

AN OBJECT-ORIENTED FRAMEWORK FOR EXPERIMENTAL CONTROL
IN THE COLTON SPIN DYNAMICS LABORATORY

by

Stephen H. Brown

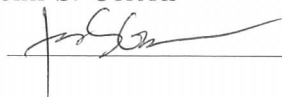
Submitted to Brigham Young University in partial fulfillment
of graduation requirements for University Honors

Department of Physics and Astronomy

August 2010

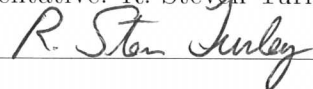
Advisor: John S. Colton

Signature: _____



Honors Representative: R. Steven Turley

Signature: _____



Copyright © 2010 Stephen H. Brown

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

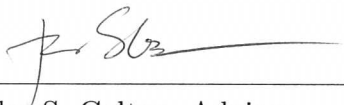
DEPARTMENT APPROVAL

of an honors thesis submitted by

Stephen H. Brown

This thesis has been reviewed by the research advisor, capstone project coordinator, and department chair and has been found to be satisfactory.

8/6/10
Date


John S. Colton, Advisor

8/9/10
Date


Robert C. Davis, Capstone Project Coordinator

10 Aug 2010
Date


Ross L. Spencer, Chair

ABSTRACT

AN OBJECT-ORIENTED FRAMEWORK FOR EXPERIMENTAL CONTROL IN THE COLTON SPIN DYNAMICS LABORATORY

Stephen H. Brown

Department of Physics and Astronomy

Bachelor of Science

This project describes computer software that helps perform spin dynamics experiments by automating data collection and instrument control in the laboratory. In this report, we review several types of spin dynamics experiments and outline their needs for computer automation. We rewrote an existing data collection application of about 15,000 lines of code using an object-oriented programming paradigm. We developed object class abstractions and interfaces that together form a unified framework for laboratory instrument automation. The new object-oriented implementation is intended to make the application more expandable, robust, and efficient. It enables new types of experiments to be performed that were not possible with the previous version of the software.

ACKNOWLEDGMENTS

This thesis would not have been possible without the help, love, and support of many people. I am grateful for the mentorship of Dr. John Colton and his limitless patience and kindness with me as I gained experience in his laboratory. Working with him has been a distinct honor and has helped me progress in my career as a scientist. I am also grateful for my many professors at BYU who taught me the useful skills I used in this project. I am also grateful for the unending love and support of my wife Tiffanie, and for the many evenings she waited patiently for me to return home from the lab. I am grateful for the love and support of my family as well.

Contents

| | |
|---|------------|
| Table of Contents | vii |
| List of Figures | xi |
| 1 Background | 1 |
| 1.1 Introduction | 1 |
| 1.2 Experiments | 2 |
| 1.2.1 Photoluminescence | 2 |
| 1.2.2 Electron Spin Resonance / Optically Detected Magnetic Resonance | 3 |
| 1.2.2.1 Initialization | 4 |
| 1.2.2.2 Spin Resonance | 6 |
| 1.2.2.3 Detection Through Kerr Rotation | 7 |
| 1.2.2.4 Detection Through Photoluminescence Polarization | 7 |
| 1.2.2.5 Analysis | 8 |
| 1.2.3 Pump-Probe Spin Lifetime Measurements | 8 |
| 1.2.3.1 Photoluminescence Polarization | 8 |
| 1.2.3.2 Time-Resolved Kerr Rotation | 10 |
| 1.3 Computer Control | 11 |
| 1.3.1 What is the Problem? | 12 |
| 1.3.2 Goals | 12 |
| 1.3.3 LabVIEW | 13 |
| 2 Primer | 15 |
| 2.1 Object | 15 |
| 2.2 Class | 16 |
| 2.3 Interface | 17 |
| 2.4 Child Classes | 18 |
| 2.5 Inheritance / Overriding | 18 |
| 2.6 Polymorphism / Dynamic Dispatch | 19 |
| 3 Methodology | 21 |
| 3.1 Object Orientation | 21 |

| | | |
|----------|---|-----------|
| 3.1.1 | Scanner and Reader Classes | 22 |
| 3.1.2 | Instrument Classes | 22 |
| 3.1.3 | Framework | 24 |
| 3.1.4 | Define Interfaces | 25 |
| 3.1.5 | Implement Polymorphism | 25 |
| 3.2 | Other New Features | 26 |
| 3.2.1 | Resource Control | 26 |
| 3.2.2 | Load/Save Functionality | 26 |
| 3.2.3 | Rewrite Instrument Drivers | 27 |
| 3.3 | Publish Code | 28 |
| 3.3.1 | Revision Control Software | 28 |
| 3.3.2 | Website | 28 |
| 4 | Interfaces | 31 |
| 4.1 | Class Hierarchy | 31 |
| 4.2 | Instrument Interface | 34 |
| 4.2.1 | Instrument.lvclass:_Destroy.vi | 34 |
| 4.2.2 | Instrument.lvclass:_InitInstrument.vi | 34 |
| 4.2.3 | Instrument.lvclass:_Refresh.vi | 35 |
| 4.2.4 | Instrument.lvclass:Check Status.vi | 35 |
| 4.2.5 | Instrument.lvclass:Control Panel.vi | 35 |
| 4.2.6 | Instrument.lvclass:Read Name.vi | 36 |
| 4.2.7 | Instrument.lvclass:Scan Setup.vi | 36 |
| 4.2.8 | Instrument.lvclass:Setup UI.vi | 36 |
| 4.2.9 | Instrument.lvclass:VISA Get Session.vi | 37 |
| 4.2.10 | Instrument.lvclass:VISA Read Async.vi | 37 |
| 4.2.11 | Instrument.lvclass:VISA Save Session.vi | 37 |
| 4.2.12 | Instrument.lvclass:VISA Write Async.vi | 38 |
| 4.3 | Readable Interface | 38 |
| 4.3.1 | Readable.lvclass:_Read.vi | 38 |
| 4.3.2 | Readable.lvclass:Graph.vi | 39 |
| 4.3.3 | Readable.lvclass:Read Channel Info.vi | 39 |
| 4.3.4 | Readable.lvclass:Read Data.vi | 39 |
| 4.3.5 | Readable.lvclass:Read Prep.vi | 39 |
| 4.4 | Scannable Interface | 40 |
| 4.4.1 | Scannable.lvclass:Get Scanner.vi | 40 |
| 4.5 | Reader Interface | 40 |
| 4.5.1 | Reader.lvclass:Reader.vi | 41 |
| 4.5.2 | Reader.lvclass:_Destroy.vi | 41 |
| 4.5.3 | Reader.lvclass:Dwell Time Delay.vi | 41 |
| 4.5.4 | Reader.lvclass:Read Dwell Time.vi | 41 |
| 4.5.5 | Reader.lvclass:Read Readable.vi | 41 |
| 4.5.6 | Reader.lvclass:Read.vi | 42 |

| | | |
|--------|---|----|
| 4.5.7 | Reader.lvclass:Scan Setup.vi | 42 |
| 4.5.8 | Reader.lvclass:Setup UI.vi | 42 |
| 4.5.9 | Reader.lvclass:Write Dwell Time.vi | 43 |
| 4.5.10 | Reader.lvclass:Write Readable.vi | 43 |
| 4.6 | Scanner Interface | 43 |
| 4.6.1 | Why Scanner and Scannable? | 43 |
| 4.6.2 | Scanner.lvclass:_Destroy.vi | 44 |
| 4.6.3 | Scanner.lvclass:_Refresh.vi | 44 |
| 4.6.4 | Scanner.lvclass:Create ArrayVarying.vi | 45 |
| 4.6.5 | Scanner.lvclass:Finish Scan.vi | 45 |
| 4.6.6 | Scanner.lvclass:First Step.vi | 45 |
| 4.6.7 | Scanner.lvclass:Has Next.vi | 45 |
| 4.6.8 | Scanner.lvclass:Next Step.vi | 46 |
| 4.6.9 | Scanner.lvclass:Range.ctl | 46 |
| 4.6.10 | Scanner.lvclass:Read Current Step.vi | 46 |
| 4.6.11 | Scanner.lvclass:Read Independent Variables.vi | 47 |
| 4.6.12 | Scanner.lvclass:Read Range.vi | 47 |
| 4.6.13 | Scanner.lvclass:Read Scannable.vi | 47 |
| 4.6.14 | Scanner.lvclass:Scan Setup.vi | 48 |
| 4.6.15 | Scanner.lvclass:Scanner.vi | 48 |
| 4.6.16 | Scanner.lvclass:Setup UI.vi | 48 |
| 4.6.17 | Scanner.lvclass:Status.vi | 48 |
| 4.6.18 | Scanner.lvclass:Write Range UI.vi | 49 |
| 4.6.19 | Scanner.lvclass:Write Range.vi | 49 |
| 4.6.20 | Scanner.lvclass:Write Scannable.vi | 49 |
| 4.7 | Scan Driver Class | 49 |
| 4.7.1 | Scan Driver.lvclass:_Destroy.vi | 50 |
| 4.7.2 | Scan Driver.lvclass:Adjust Min-Max.vi | 50 |
| 4.7.3 | Scan Driver.lvclass:Build Scan Graph.vi | 50 |
| 4.7.4 | Scan Driver.lvclass:Create Data Store.vi | 50 |
| 4.7.5 | Scan Driver.lvclass:Estimate Time.vi | 50 |
| 4.7.6 | Scan Driver.lvclass:Factory.vi | 51 |
| 4.7.7 | Scan Driver.lvclass:Finish Scan.vi | 51 |
| 4.7.8 | Scan Driver.lvclass:Load Scan.vi | 51 |
| 4.7.9 | Scan Driver.lvclass:Pre-Scan Check.vi | 52 |
| 4.7.10 | Scan Driver.lvclass:Scan Driver (for Setup UI).vi | 52 |
| 4.7.11 | Scan Driver.lvclass:Scan.vi | 52 |
| 4.7.12 | Scan Driver.lvclass:Settings Panel.vi | 52 |
| 4.7.13 | Scan Driver.lvclass:Setup.vi | 53 |
| 4.8 | Creating New Objects | 53 |

| | |
|---|-----------|
| 5 Applications | 55 |
| 5.1 Etch-A-Sketch | 55 |
| 5.2 Scan Driver | 56 |
| 5.2.1 How to Expand Scan Driver | 60 |
| 6 Conclusion | 63 |
| Bibliography | 65 |
| Glossary | 67 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Electron spin state transitions in GaAs. | 5 |
| 1.2 | GaAs electron spin resonance detected via Kerr rotation. | 9 |
| 1.3 | Bloch spheres, with an external magnetic field pointing up. | 9 |
| 1.4 | Pump-probe laser beam. | 9 |
| 2.1 | Car class definition. | 16 |
| 5.1 | Etch-A-Sketch front panel. | 56 |
| 5.2 | Constructing Objects in Etch-A-Sketch. | 57 |
| 5.3 | Main read loop in Etch-A-Sketch. | 57 |
| 5.4 | Scan Driver constructor front panel. | 58 |
| 5.5 | Scan Driver constructor block diagram. | 58 |
| 5.6 | Scan Driver setup UI front panel. | 58 |
| 5.7 | Setup UI block diagram. | 59 |
| 5.8 | Scan method front panel. | 60 |
| 5.9 | Scan method main loop block diagram. | 61 |

Chapter 1

Background

1.1 Introduction

The burgeoning field of spintronics promises to vastly improve our computer and communication systems by exploiting the spin properties of the electron. However, great challenges remain to control, maintain, and measure the spin states of spintronic systems. Much research is devoted to finding ever better ways to maintain coherence of spin states and reliably control and probe the spin states of these systems. Due to the short timescales involved in spin-state manipulation and the fragility of the spintronic system, precise computer control of equipment is crucial for successful research.

The Colton Lab performs spintronics experiments on semiconductors such as gallium arsenide (GaAs), and these experiments fall into several basic categories. One type of experiment, photoluminescence, excites the electrons in the semiconductor to reveal optical transitions. Another experiment, electron spin resonance, induces transitions that reveal the lifetime of spin states as well as the semiconductor's g-factor. Further, pump-probe experiments alternately inject and detect spin states, which can

also reveal the lifetime of the spin states in the semiconductor. What follows is a brief description of the theory and practice of each of these experiments.

1.2 Experiments

1.2.1 Photoluminescence

In a photoluminescence experiment, a laser beam shines on a semiconductor sample to induce its VALENCE BAND¹ electrons to transition to a higher energy level in its CONDUCTION BAND. The electrons will eventually transition back down to the valence band, emitting photons with energies that make up for the difference between the two energy levels. By collecting and studying this PHOTOLUMINESCENCE, one can deduce the energy states within the material.

A spectrometer uses a diffraction grating to separate the photoluminescence into its constituent wavelength bands. Then, by varying which wavelength reaches the output of the spectrometer, and recording the light intensity at that output with a detector, one can create a graph of the spectrum of the emitted light.

A computer can automate this experiment by controlling a spectrometer's wavelength while collecting data from the detector, which could be, for example, a photodiode connected to a lock-in amplifier or a photomultiplier tube connected to a photon counter.

¹Throughout this document, terms in SMALL CAPS are defined in the Glossary on page 67.

1.2.2 Electron Spin Resonance / Optically Detected Magnetic Resonance

Electrons and photons carry quantized units of angular momentum called SPIN; quantum mechanics restricts these quanta to \pm half-integer units for electrons, and ± 1 unit for photons. A photon's spin determines whether its optical circular polarization is right-handed or left-handed. Since nature requires that all interactions conserve angular momentum, if we know the spin state of a system before an electron-photon interaction, we can determine its spin state afterward as well.

In most bulk III-V semiconductors, such as GaAs, electrons in the top of the valence band have spins of $-\frac{3}{2}$, $-\frac{1}{2}$, $\frac{1}{2}$, or $\frac{3}{2}$. Conversely, electrons in the bottom of the conduction band have spins of $-\frac{1}{2}$ or $\frac{1}{2}$.² These available spin states determine the possible transitions. For example, when a photon with spin $+1$ excites an electron in the $-\frac{3}{2}_{VB}$ state, the electron can go to the $-\frac{1}{2}_{CB}$ state; if the electron is in the $-\frac{1}{2}_{VB}$ state, it can go to the $\frac{1}{2}_{CB}$ state.

One might think that the electrons fill in the $-\frac{1}{2}_{CB}$ and $\frac{1}{2}_{CB}$ states in equal amounts. However, due to the nature of the material, the $-\frac{3}{2}_{VB} \mapsto -\frac{1}{2}_{CB}$ transition is much more likely than the $-\frac{1}{2}_{VB} \mapsto \frac{1}{2}_{CB}$ transition by a 3:1 ratio (see figure 1.1a). By exciting the material in this way, one can create an excess POPULATION of electrons in the $-\frac{1}{2}_{CB}$ state. When many of a material's electrons are in the same spin state, we call it SPIN-POLARIZED (not to be confused with optical polarization of light).

In the reverse case, when an electron drops down from the conduction to the valence band, it emits a photon with spin -1 or $+1$, which is right- or left-circularly polarized light (see figure 1.1b). The same 3:1 ratio applies, so electrons in the $-\frac{1}{2}_{CB}$

²It is assumed for the sake of discussion that $-\frac{1}{2}_{CB}$ is a lower energy state than $\frac{1}{2}_{CB}$, although in some materials the reverse is true.

state will be more likely to drop down to the $-\frac{3}{2}V_B$ state (emitting a +1 photon) than to the $\frac{1}{2}V_B$ state (emitting a -1 photon). Thus, this PHOTOLUMINESCENCE POLARIZATION indicates the spin states of the excited electrons.

Normally, electrons in different spin states have identical DEGENERATE energies. However, when a paramagnetic material (like GaAs) is placed in a uniform magnetic field, the energies of its electrons' spin states split into distinct levels. The Zeeman effect describes the energy difference E between these spin states:

$$E = g\mu_B B \quad (1.1)$$

where g (the “g-factor”) is a constant that depends on the material, μ_B is the Bohr magneton, and B is the applied magnetic field's strength.

At typical laboratory fields (1 T) the energy levels for GaAs differ by $f = \frac{E}{h} \approx 6$ GHz, which is in the microwave band. The basis of an ELECTRON SPIN RESONANCE (ESR) experiment is to excite electrons into the conduction band, induce transitions between conduction band spin states by applying the requisite microwave energy, and to observe the effects (see figure 1.1c). This enables a scientist to control and study the spin states of the material [1].

1.2.2.1 Initialization

The materials being studied contain extra (“doped”) electrons, so that there are electrons in or near the CB even in the absence of light. The first step in the experiment is to initialize all (or, in practice, just an observable fraction) of the doped electrons into the same spin state in the conduction band. There are two ways to do this. The first is through OPTICAL PUMPING with a circularly polarized laser. The laser can induce transitions from the valence band to the conduction band and populates electrons in the $-\frac{1}{2}CB$ state (see figure 1.1a). If an appreciable number of spin-polarized

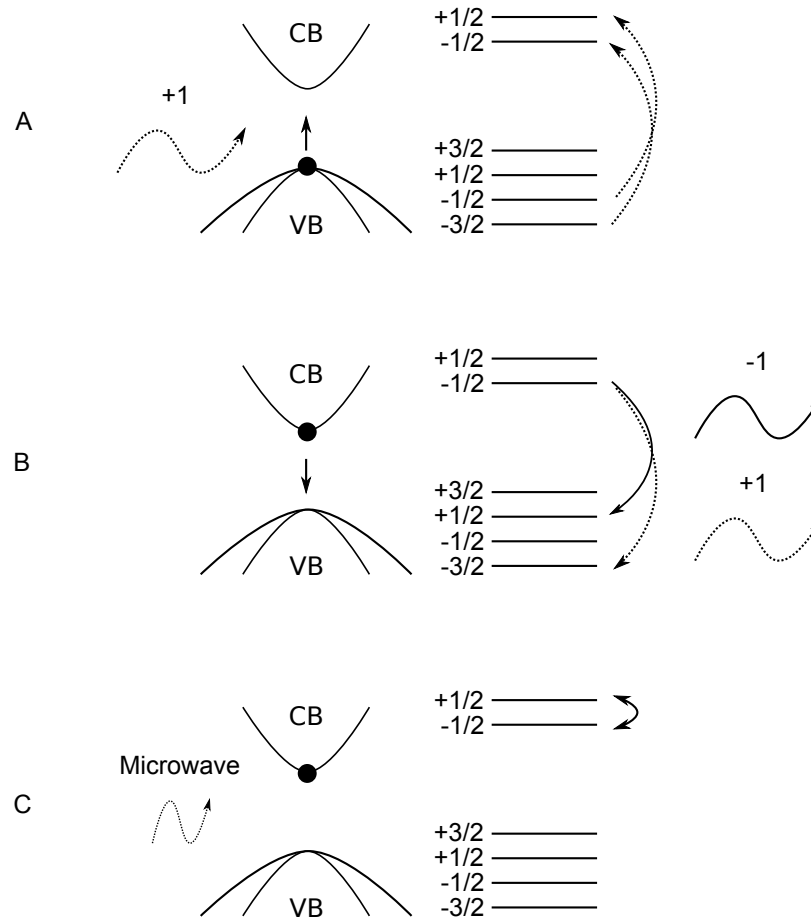


Figure 1.1 Electron spin state transitions in GaAs. A magnetic field splits the valence band (VB) and conduction band (CB) into spin-related energy levels.

(a) Photons at optical frequencies can excite electrons from the VB into the CB. In GaAs, the $-\frac{3}{2}_{VB} \mapsto -\frac{1}{2}_{CB}$ transition is preferred when photons with spin $+1$ are used.

(b) CB electrons drop down to the VB, emitting optical photons with spin $+1$ or -1 . If electrons are spin-polarized in the $-\frac{1}{2}_{CB}$ state, only two transitions are possible. In GaAs, the $-\frac{1}{2}_{CB} \mapsto -\frac{3}{2}_{VB}$ transition is preferred.

(c) Photons at microwave frequencies induce transitions between CB spin states.

electrons are optically injected into the CB, they interact with and spin-polarize the doped electrons via “spin exchange.”

The second method to initialize the spin states is through THERMAL EFFECTS. When the material is cooled down to very low temperatures (≈ 1.5 K) and placed in a strong enough magnetic field (≥ 1 T for GaAs), ambient thermal energy ($k_B T$) in the material is small enough that it is comparable to the spin splitting energy ($g\mu_B B$). Under these conditions, electrons already in the conduction band begin to accumulate in the $-\frac{1}{2}CB$ state. While optical pumping can be used to populate electrons into either CB spin state, thermal effects can only be used to initialize the electrons into the lower energy state.

1.2.2.2 Spin Resonance

After the spin states have been initialized to have an excess population in the desired spin state, application of microwave energy (at the correct resonant frequency) will induce transitions between the two spin states and cause the spin populations to even out (see figure 1.1c).

Since this resonant frequency depends on the magnetic field B , resonance can be attained by either varying the frequency while holding the magnetic field constant, or varying the field while holding the frequency constant. Typically the latter method is preferred, as resonant microwave cavities are frequently employed to strengthen the microwave field for one particular frequency.

Detecting and comparing the spin populations at the initialization and resonance phases will indicate if the resonance condition has been achieved.

1.2.2.3 Detection Through Kerr Rotation

The magneto-optical Kerr effect describes the change in optical polarization of a reflected beam of light when it reflects from a magnetized material's surface. Because the spin-polarization of a material behaves like a magnetization, the Kerr effect enables one to detect the spin states of a material [2].

In this detection scheme, a probe laser beam, linearly polarized, reflects from the sample while the magnetic field is varied, and a detector notes the beam's changing polarization due to the Kerr effect. The change in the laser's optical polarization indicates a change in the surface magnetization, and hence spin-polarization, of the material.

In an example experimental setup, the microwave energy turns on and off at a fixed interval to provide a reference signal. The reflected laser beam passes through a polarizing beam splitter, and a balanced detector compares its polarization components. A lock-in amplifier, referenced to the "chopped" microwave signal, then takes data from this detector.

Reference [3] describes an experiment that used thermal polarization and Kerr rotation detection to perform electron spin resonance.

1.2.2.4 Detection Through Photoluminescence Polarization

As described above, if a laser optically injects spin-polarized electrons into the CB, the doped electrons become spin-polarized via spin exchange. Then, the electrons will emit *circularly* polarized light as they drop down to lower VB states (see figure 1.1b). However, after the electrons are spin-polarized into one state, the microwave energy causes resonance that destroys this spin-polarization. Thus the spin-polarization of the material can be monitored by measuring the photoluminescence polarization produced in response to a probe beam [1].

In this experiment, a circularly polarized laser beam (at a fixed wavelength) excites the material while the magnetic field is varied, and a detector measures the polarization of light emitted from the material. Any decrease in photoluminescence polarization indicates spin resonance has been achieved.

In both detection methods, a computer can automate the magnetic field sweep while collecting data from the detector, typically a photon counter acting synchronously with an optical retarder.

References [5] and [6] describe experiments that used optical pumping and photoluminescence polarization to perform electron spin resonance.

1.2.2.5 Analysis

Two quantities of interest to research are immediately evident from a graph of relative spin-polarization versus magnetic field strength (see figure 1.2). The magnetic field at the peak on this graph indicates the material's g-factor:

$$g = \frac{h\nu_{microwave}}{\mu_B B_{resonant}} \quad (1.2)$$

The energy-time uncertainty principle, $\Delta E \Delta t \geq \frac{\hbar}{2}$, implies that the half-width of this peak (at half-maximum) limits the relative lifetime " T_2^* " of the spin states [5]:

$$(T_2^*)^{-1} \leq \frac{g\mu_B \Delta B_{hwhm}}{\hbar} \quad (1.3)$$

1.2.3 Pump-Probe Spin Lifetime Measurements

1.2.3.1 Photoluminescence Polarization

There are many ways to measure the lifetime of a system of spin-polarized electrons. In this experiment, a semiconductor sample is cooled down to liquid helium temperatures and placed in a fixed magnetic field so its spin states split into two distinct energy levels.

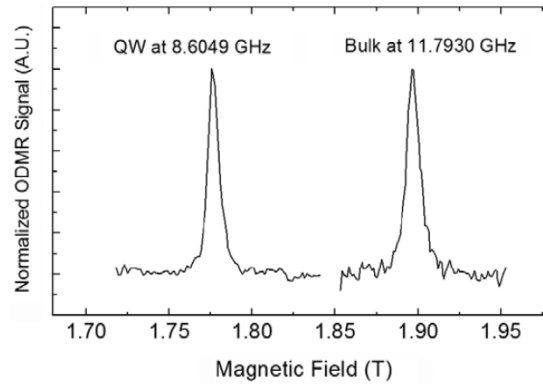


Figure 1.2 GaAs electron spin resonance detected via Kerr rotation, from reference [3]. Two materials are shown: a quantum well (QW) sample and a bulk GaAs semiconductor. The optically-detected magnetic resonance (ODMR) signal indicates the relative spin-polarization as a function of magnetic field and the denoted microwave frequency. The differing resonant peaks demonstrate the different g-factors of the two materials.

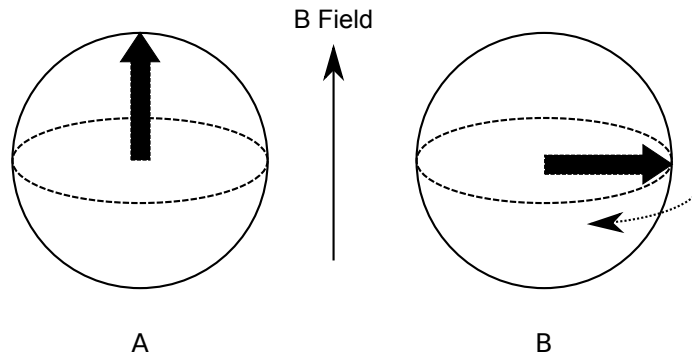


Figure 1.3 Bloch spheres, with an external magnetic field pointing up. (a) Spin aligned parallel to field. (b) Spin aligned perpendicular to field.

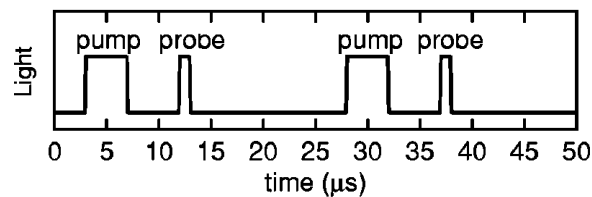


Figure 1.4 Pump-probe laser beam, from reference [4].

A circularly polarized laser, pulsed at specific intervals, alternately injects and detects spin-polarization in the material. A “pump” laser pulse first spin-polarizes the doped electrons *parallel* to the magnetic field (see figure 1.3a).

Then another “probe” pulse excites the electrons, after which they drop down and emit circularly polarized light. These pulses are produced electronically from a single laser beam (using, e.g., an optical modulator) so that the spacing between pulses can be on the order of microseconds (see figure 1.4). Since some electrons lose their spin-polarization during the delay between the pump and probe pulses, one can increase this delay until the probe pulse detects little remaining spin-polarization. This technique can reveal the longitudinal lifetime “ T_1 ” of the spin states.

In this experiment, the delay between pump and probe pulses is varied and a detector records the emissions. As explained in section 1.2.2.4, the emitted light indicates the spin-polarization of the material; how much remains reveals the spin decay lifetime of the material. A computer can automate this experiment by controlling the laser pulses while recording data from a detector.

1.2.3.2 Time-Resolved Kerr Rotation

Another pump-probe experiment is TIME-RESOLVED KERR ROTATION. In this experiment, again the sample is cooled and placed in a fixed magnetic field. This time, a circularly polarized pump pulse aligns the spins *perpendicular* to the magnetic field (see figure 1.3b). In this way, the electrons are not in eigenstates, and instead their spins precess about the axis of the magnetic field.

In this method, a single pulsed laser beam is used, split into two unequal halves. The first beam pumps the material with circularly polarized light pulses. Since pulsed lasers typically cannot have arbitrary pulse intervals, a mechanical “translation stage” forces the second pulse to travel a longer distance, and thus it arrives at the sample

after the first pulse. In this way, one can obtain arbitrary pump-probe delay times on the order of nanoseconds.

The second beam probes the spin-polarization with *linearly* polarized light pulses. This enables detection of the transverse spin states by the same method of Kerr rotation as described above. Again, with a longer pump-probe pulse interval, more electrons lose their spin-polarization.

In this experiment, the translation stage moves to vary the pulse interval while a detector records the reflected probe pulse's change in optical polarization due to Kerr rotation. This indicates the spin-polarization left in the material. A computer can automate the movement of the translation stage while recording the polarization of the reflected laser beam.

Correlating the pump-probe delay with this relative spin-polarization produces an oscillating graph that exponentially decays. The period of oscillation indicates the material's g-factor, and the time constant of the decay curve envelope indicates the transverse lifetime " T_2^* " of its spin states.

1.3 Computer Control

BYU's spin dynamics research laboratory, under the direction of Dr. John Colton, employs a LabVIEW application platform to collect data from multiple sources and automate control of laboratory instruments during the aforementioned experiments. This software, called the COLTON LAB SCAN SOFTWARE, comprises the equivalent of over 15,000 lines of source code and is a critical tool in the study of spin dynamics.

1.3.1 What is the Problem?

The platform has undergone many iterations and bug fixes since its creation in 2005. The first version of the software was intended to monitor a photon counter while sweeping over a magnetic field. It was very limited in scope and intended for temporary use.

This code gradually expanded beyond its original purpose to perform many other types of experiments as the need arose. New code was added over the years in an ad-hoc manner without addressing its design. The code base gradually entered a state of “spaghetti code” since it was being adapted far beyond its original intent.

The design of the original code base had become so cluttered that it became harder and harder to add features. In particular, the original software had no capability to perform time-resolved Kerr rotation (section 1.2.3.2), and the search for a way to seamlessly integrate this type of experiment revealed that rewriting the code base was the only feasible way to proceed.

1.3.2 Goals

The purpose of this project was to “clean up” the code base using a planned design and widely accepted programming conventions. The process aimed to create component abstractions (such as generic handlers for arbitrary instruments), develop a framework, and rewrite the entire code base on top of this framework. It was also necessary to avoid major changes all at once so as not to disrupt daily operations.

Carefully rewriting the code base had three major benefits. Largely motivating this project was its ability to enable experiments not possible with the original software, such as time-resolved Kerr rotation (section 1.2.3.2), as well as many others. It enabled the addition of many new features and conveniences (section 3.2). Hopefully,

it will also ensure the software remains maintainable and expandable for many more years.

1.3.3 LabVIEW

LabVIEW is a commercial C-like programming language coupled with a rapid application development environment. Its graphical “G” programming language and intuitive integrated development environment let novice programmers quickly begin developing software. Its internal compiler performs comparably to C and can target many architectures, including desktops, mobile devices, and FPGAs.

LabVIEW is widely used in science and engineering for data acquisition and instrument automation, using interfaces such as IEEE-488 “GPIB” that are standard on most industrial laboratory equipment. Scientific equipment vendors commonly supply LabVIEW drivers with their equipment – and perhaps *only* LabVIEW drivers. LabVIEW is also surrounded by a healthy online community of enthusiasts and libraries.

LabVIEW has existed in some form since its 1986 debut on the Apple Macintosh. Its compiler remains backwards-compatible with all prior versions of the suite, and so a wealth of “ancient” applications and libraries can run essentially unmodified. In 2006, LabVIEW gained support for object-oriented code, and, in 2009, support for recursive function calls. These new features give more credibility to LabVIEW’s claim to be a general purpose programming language. They also enable serious software development using widely accepted programming practices, which is the intent of this project.

Chapter 2

Primer

Computers are extremely complex machines, and managing this complexity is the key to creating reliable systems. Several solutions exist to clean up and organize programming code, and one powerful method is object orientation. Object orientation can simplify the design and increase the reliability of a program, and is well-suited for application to this software.

Many proposed enhancements (chapter 3) involve object orientation of the existing code, so a review of some basic concepts from object-orientation theory is described in the following pages. For a more thorough explanation, please consult any good computer science textbook; some references are given in [7–9].

2.1 Object

An OBJECT is an imaginary representation of data inside a computer program. A car, a bike, or a scooter could be an object. A computer program can use these objects through their METHODS - functions of code that manipulate an object's data. Depending on its type, an object might have a method called `car.drive()`, `bike.pedal()`,

| Member Fields | | Private Methods | Public Methods (Interface) |
|------------------------|--------------------------|--------------------------------|------------------------------------|
| <code>Car.color</code> | <code>Car.model</code> | <code>Car::ignition()</code> | <code>Car::driveTo(Address)</code> |
| <code>Car.year</code> | <code>Car.mileage</code> | <code>Car::shiftGears()</code> | <code>Car::stop()</code> |
| <code>Car.make</code> | <code>Car.speed</code> | | |

Figure 2.1 Car class definition. A Car class contains several member fields and methods. Outsiders may call public methods to accomplish tasks, while private methods, for internal use only, take care of the details.

or `scooter.scoot()`. A whole fleet of Car objects could be stored inside an array, and a program could drive them all at once.

2.2 Class

A CLASS defines an object in abstract terms. The class is the object's abstract definition, and the object is the data in memory (INSTANCE) that fits the definition. A Car class defines FIELDS – properties of all Cars, e.g. their `color`, `year`, `make`, and `model` – as well as the object's methods (see figure 2.1).

For example, a Car object stored in the variable `alice's_car` is a White 1998 Saturn SL1, and a different Car object in the variable `bob's_car` is a Grey 2001 Ford Escort. These two objects are both instances of the Car class. Note that only object instances have physical presence in the computer's memory – classes only exist in theory to define the objects.

Internal to the computer, DATA STRUCTURES define class fields, and structured memory holds the objects for use. In LabVIEW, a data structure is a "cluster" data type; thus, a class is a specific type of cluster, and object instances "live inside" those clusters.

2.3 Interface

An `INTERFACE` defines the way a class can interact with other classes. Specifically, the interface consists of the class's `PUBLIC METHODS`: all the methods a class allows others to use.

The class may contain other `PRIVATE METHODS` that outsiders may not call – these may cause unpredictable behavior and are marked for internal use only (see figure 2.1). Generally, class fields are also marked as private. The distinction between public and private methods to create interfaces is a type of `ENCAPSULATION`: a way of hiding internal complexity to make software components more robust and easier to integrate.

There are generic interfaces and specific interfaces. One class may have its own `SPECIFIC INTERFACE` for its own specialized purpose. As an analogy, the jumble of wires running from a breadboard to an experiment defines a custom interface; the experimenter will know what these wires are for, but others will not.

A `GENERIC INTERFACE` can be used by more than one class as long as each class implements the same public methods. An analogy is the standard AC electrical socket, which contains hot, neutral, and ground wires. Any appliance with this interface can use the wall's power supply. Generic interfaces are powerful because they make programs modular: one can snap in and out components at will, because their interfaces "fit" all the same. Imagine if houses did not have standardized power plugs – what a mess that would be!

For example, in the course of rewriting the Colton Lab Scan Software, we invented the `Instrument` interface, which defines behavior common to all instruments in the lab. When other classes `IMPLEMENT` this interface's methods, they too become an `Instrument`, and can be used anywhere in the program that an `Instrument` can. The

`Instrument` interface has these methods, outlined in detail in section 4.2:

| | |
|---------------|-----------------|
| Check Status | Init Instrument |
| Control Panel | Read Name |
| Destroy | Setup UI |

To ready the software to use a new piece of equipment, one needs only to create a new `Instrument` object by defining these methods (see section 4.8). Few other parts of the software need modification.

2.4 Child Classes

Classes may have `CHILD CLASSES` that define new and additional behavior. For example, a `Vehicle` class may have these children classes: `Car`, `Bike`, `Scooter`. The `Vehicle` class defines the methods common to all types of vehicles, while child classes define behavior specific to their own kind. For example, a car could honk its horn, but a `Bike` and `Scooter` would ring a bell, and the `Vehicle` class could not generalize this behavior.

In LabVIEW, to implement an interface, one must define a child class of that interface.

2.5 Inheritance / Overriding

If a child class so desires, it may `OVERRIDE` its parent's default methods to define new behavior. For example, a `Vehicle` class may honk a generic horn, but a `Car` class could override this to sound a specific car's horn, while a `Bike` class could override this to sound a bell. Child classes can modify and adapt an existing class by overriding instead of permanently changing the parent class's methods.

`INHERITANCE` is just the opposite of overriding. If a child class does not override

its parent's methods to specify different behavior, the compiler automatically reuses the parent's default methods.¹ This property lends itself to reduction of duplicate code: many common blocks of code can be consolidated into a single parent class method, with idiosyncrasies handled by overrides within child classes.

Coupled together, inheritance and overriding from parent classes allow one to quickly re-purpose existing code for new situations, as well as cut down on duplicate code that is difficult to maintain.

2.6 Polymorphism / Dynamic Dispatch

A program may contain many blocks of code like this, which discerns the type of an object in order to choose the appropriate task:

```
if (instrument == 1) {
    instrumentOneFunction(); //do something
} else if (instrument == 2) {
    instrumentTwoFunction(); //do something else
} else if (instrument == 3) {
    . . .
```

In case the user wants to add a new instrument, s/he must locate every section of the code that looks like this and update it accordingly. This is difficult to maintain, often gets overlooked or forgotten, and causes many bugs.

POLYMORPHISM is a compiler construct that automatically chooses the method appropriate for the object when the program runs. Polymorphism can eliminate the

¹Compare the 10th Amendment to the U.S. Constitution: "The powers not delegated to the United States by the Constitution, nor prohibited by it to the States, are reserved to the States respectively, or to the people."

above problem by turning it into the following. Notice that this code does not need to be updated to handle new classes:

```
instrument.function(); //a polymorphic method
```

A method called at run-time in this fashion is sometimes called a DYNAMICALLY DISPATCHED method.

Polymorphism is subtly distinct from overriding. The specific distinctions are beyond the scope of this document, but suffice it to say that polymorphism's internal compiler constructs allow one to *refer* to a group of objects in a generic way, while still maintaining their idiosyncrasies. Overriding alone does not permit reference to many objects in a generic way.

Chapter 3

Methodology

“Cleaning up” a software project leaves much to interpretation. Here are some of the specific techniques used to redesign the software. These techniques are common practice in software engineering; some references are given in [7–9].

3.1 Object Orientation

The Colton Lab Scan Software performs a vast array of instrument automation and data collection functions. Many of its routines are strongly intertwined and difficult to debug. Object orientation can help solve this problem by decomposing the program into independent parts and giving those parts well-defined roles as objects with methods.

Object orientation is the process of designing a program’s functionality in terms of objects. This necessitates describing exactly what a program is supposed to do, then breaking that purpose down into components that help accomplish the goal. The program’s components can often be constructed from one or more objects, which carry out the actual tasks of the program. This process, called a TOP-DOWN DECOM-

POSITION, reveals the requisite objects and what functionality each of them needs.

The Colton Lab Scan Software is designed to control one or more instruments and automate data collection, as described in section 1.2. The following pages describe the components that accomplish this task, which were invented during the course of this project.

3.1.1 Scanner and Reader Classes

The primary application of the Colton Lab Scan Software is the **Scan Driver**. Since the **Scan Driver** ultimately produces an x-y graph, it makes sense to define two components responsible for the x and y parts. We will call the controller for the x (“independent”) variable a **Scanner**. We will call the controller for the y (“dependent”) variable a **Reader**.

Scanners are responsible to set the independent variable, e.g. a magnetic field or spectrometer as outlined in section 1.2. **Readers** are responsible to read in the dependent variable from a data source, such as a photon counter or lock-in amplifier.

Preprocessing of the data (such as computation of spin-polarization from the two channels of the photon counter) generates an additional data source not handled by the **Reader**. We will delegate an additional class called **Aux Data Source** responsible for these tasks. This class is analogous to **stream** operators in other languages.

The full implementation details of these classes are given in sections 4.5 and 4.6.

3.1.2 Instrument Classes

Scanners and **Readers** are simply abstract representations of the experiment. They need to interact with equipment to perform their task. It makes sense to classify laboratory equipment in generic terms so that **Scanners** and **Readers** may interact

with all equipment.

The simplest class is **Instrument**, which provides functionality common to all computer-controlled equipment. Obviously, this limits the class to very few tasks, like the universal **Check Power On** and generic functions like **Get Status**.

Some **Instruments** can be grouped into subclasses. If an **Instrument** is designed to change position, we will call it **Scannable**. **Scannable** objects include a spectrometer, electromagnet, pulse generator, laser power controller, cryostat temperature controller, microwave function generator, etc.

Similarly, if an **Instrument** is primarily a data source, we will call it **Readable**. **Readable** objects include a Photon counter, Lock-in amplifier, A-D converter, etc.

Some instruments do not fit into either category (such as an LCD retarder), and remain **Instruments**. Moreover, some instruments fit into both categories (such as a temperature controller). To classify an **Instrument** as both **Scannable** and **Readable** requires **MULTIPLE INHERITANCE**, a feature which LabVIEW does not support as of this writing. In this case, one must create both a **Scannable** and **Readable Instrument** for the same physical piece of equipment.

These classes were chosen to dovetail nicely with the **Scanner** and **Reader** classes described above. Routines within the main application that control **Readable Instruments** are **Readers**, and routines that control **Scannable Instruments** are **Scanners**. The full details of the **Scannable** and **Readable** classes are given in section 4.4 and 4.3.

3.1.3 Framework

To summarize, all elements of the system fit into one of the following ABSTRACTION LAYERS:

1. Main application launcher/dashboard
2. `Scan Driver` program; other applications such as `Etch-A-Sketch` (sections 3.1.1, 5.1, and 5.2)
3. `Scanner`, `Reader`, `Aux Data Source` objects classes (section 3.1.1)
4. Object-oriented (“molecular”) `Instrument` classes (section 3.1.2)
5. Static (“atomic”) instrument drivers (section 3.2.3)
6. Low-level bus communication (section 3.2.3)
7. Hardware (section 3.2.3)

Together these abstraction layers comprise a FRAMEWORK. In this framework, code at one abstraction layer can only communicate with code one layer above and one layer below it. This is called STRONG LAYERING.

One benefit of the aforementioned object classes is reusability – they can serve multiple purposes. Specifically, the `Instrument` classes and subclasses are not limited to use in a `Scan Driver` application (section 5.2); they can be instantly reused to build other applications as well. A well-designed class hierarchy will permit this flexibility and enable future growth.

An example of reusability in action is that multiple applications can use `Instrument` objects. `Etch-A-Sketch` (section 5.1) is a program that reads a data source and plots it on the screen in real time, much like an oscilloscope. It can use the `Readable` objects that `Scan Driver` uses without modification.

3.1.4 Define Interfaces

The previous application platform was very strongly coupled: changing one system of the program would affect many unrelated systems. For example, one routine for the pulse/pattern generator called an internal function of the photon counter drivers. This coupling is poor design.

The object classes defined above have public and private methods (see section 2.3), prohibiting interaction between objects except at predefined interfaces. This keeps the objects independent of each other and ensures they have well-defined roles.

3.1.5 Implement Polymorphism

Polymorphism, outlined in section 2.6, eliminates bugs by using a compiler construct to automatically call the appropriate function for a given object. We have implemented polymorphism in our object hierarchy.

The previous software had six duplicate copies of its internal scan loop, for many combinations of instrument configurations. The software did not allow the user to flexibly configure instruments at the start; the loop for each specific instrument combination was hard-coded into the program. This unnecessary duplication of code created bugs, because changing one variant of the loop did not update the other copies.

Polymorphism eliminates the six duplicated scan loops and combines them all into a single loop, while allowing the user to configure equipment dynamically. This enables the application to perform the time-resolved Kerr rotation experiment (section 1.2.3.2), which was impossible in the previous version of the software. Moreover, it is not limited to preconceived experiments: polymorphism enables the user to quickly configure any combination of equipment for an experiment, without any modification

to the software.

3.2 Other New Features

3.2.1 Resource Control

LabVIEW is an inherently parallel language, and sometimes threads in the scan software inadvertently compete for access to peripheral equipment. When two parts of the software attempt to access the same equipment at the same time, this creates bus contention errors.

In the process of abstracting equipment drivers into a class hierarchy, this project implemented MUTEX (mutual exclusion) resource locks on shared equipment, such that one thread cannot access the equipment until another thread releases its lock. A mutex is like a key to the restroom: only one person can have the key at a time.

3.2.2 Load/Save Functionality

In science, it is critical to be able repeat an experiment exactly as before. Careful records and auditing assist in this goal. Adding load/save functionality protects the integrity of results; it also enables the software to call up a previous scan and repeat it later using the same settings.

The previous application platform used a large global hash table to store instrument settings. This was difficult to write to disk and load from a saved file, since the file writer and parser needed to handle each entry in the hash table. Structured data within object classes make load/save capabilities easier to manage.

Since object classes include member fields, it makes sense to store instrument settings inside the member fields. One object can use another, and even store an-

other object inside its member fields. For example, a `Scanner` object may store the `Scannable` instrument with which it is associated for easy access. This hierarchy of objects within member fields creates a nested `DATA STRUCTURE` which groups settings logically, instead of storing them all inside a large hash table.

A `TREE` like this is easy to `SERIALIZE`, or write from memory to disk. Data structures facilitate an easy way to load and save scan settings to and from disk.

3.2.3 Rewrite Instrument Drivers

The main program's object classes call static instrument drivers (abstraction level 5 from section 3.1.3) which communicate directly with the instruments. These drivers are simple functions designed to send one specific GPIB command. They use lower-level system routines (abstraction level 6) to communicate with the instruments (abstraction level 7).

Many of these drivers needed revision. These drivers were either provided by manufacturers or written by students from technical documentation. Many were unreliable and in some cases did not follow the equipment manufacturers' specifications.

Each instrument now has two sets of drivers. First, static drivers – those that do not store any state – contain the simple “atomic” commands, e.g. “Set Wavelength to XX nm.” These perform one task and do it well.

Second, Object-oriented drivers combine “atomic” commands to perform larger “molecular” instrument tasks, e.g. set up for scan, that can be handled in a generic way common to every instrument. The Object-oriented drivers all implement an interface common to all instruments. In this way, generic scan programs can call the class-based drivers, which call the deeper atomic commands.

Separating the objects from the static drivers gives the benefits of polymorphism and interfaces (unique to objects) while retaining the simple architecture of static

drivers. This also separates the instrument driver code from the code specific to our lab, making it possible for others to reuse the static instrument drivers we have written.

3.3 Publish Code

3.3.1 Revision Control Software

Revision control, a system that backs up all prior versions of a file, is an absolute necessity for large code bases. Previously, a script backed up the source code by copying the source tree to folders named by the current date.

The industry standard Subversion revision control system is the system of choice. It stores data efficiently, includes a change log, and is more flexible than scripted backups. We created a way to integrate Subversion revision control with the LabVIEW development environment. Then we overlaid the Subversion system onto the source code tree and trained the students in the lab to use it properly.

3.3.2 Website

This project is intended to be as generic and re-usable as possible. For example, even though it required extra work, we provided two sets of instrument drivers, one for internal use and one for redistribution. In addition, the class abstractions enable the software to be quickly repurposed for new experiments, with few modifications. Thus the design and many of its components have definite value to the outside world and can be easily reused by other groups.

Subversion software, combined with a well-known license, makes it a trivial task to offer the code to others. This lets others benefit from the existing work, and gives

our efforts the maximum impact. Publishing software to the open Internet also boosts BYU's worldwide reputation and visibility, especially among programming circles.

This work has been published as Open Source software, under the GNU General Public License (version 3) so that others are free to re-use and modify it for their own needs. As of this writing, all code is available on the Google Code website, <http://coltonlab.googlecode.com>, under this license.

Chapter 4

Interfaces

This section outlines the specifications for the public methods of the major object classes described above, current as of this writing. As with all software projects, the *canonical* specification of the object classes is the *code itself*. The following documentation is for reference purposes only. As of this writing, the latest code is available online at <http://coltonlab.googlecode.com>.

4.1 Class Hierarchy

The full class hierarchy is as follows. Indentation indicates subclass dependence.

- class InstrumentPointer // a wrapper class used to put Instrument objects on the heap
- class Instrument //generic interface; implements init(), controlPanel(), reset(), checkOn()
 - class LCDRetarder
 - class Readable //generic Instrument interface; implements read()

- * class DAQmxAnalog
- * class FunctionGenerator
- * class LockinSR830
- * class PhotonCounter
- * class ReadableUtility //pseudo-classes not associated with a physical instrument
 - class Collection //container class: an array of Readables
 - class Dummy //inert Readable Instrument for testing
 - class Minus //displays the difference between two channels of a Readable
 - class StdDev //calculates time-averaged standard deviation of this source
 - class Timer // read the experiment's elapsed time as a source
- class Scannable //generic Instrument interface; implements getScanner()
 - * class Magnet
 - * class Microwave
 - * class PulseGenerator
 - * class Spectrometer
 - * class Stepper
- class FileIO // File & disk routines, laboratory database access, object serialization routines.
- class ScanDriver // a generic, instrument-agnostic data collector that operates on Objects.

- class Scanner //generic Scannable iterator, with methods getFirst(), getNext(), hasNext()
 - class BfieldScan // scans over the Scannable::Magnet at specified field points
 - class BfieldScanContinuous // scans over Scannable::Magnet at a continuous interval
 - class RFScan //scans over Function & Microwave generators, sweeping fields
 - class PulseWidthScan //scans over PulseGenerator, varying width of pulses
 - class WaitTimeScan // scans over PulseGenerator, varying inter-pulse delay
 - class WavelengthScan //scans over an interval on the Spectrometer
 - class PulseGenScan //varies the pulse width or pulse delay of the Pulse Generator
 - class TranslationStage //steps a translation stage object
 - class VarianMagnet //sweeps the field of a Varian magnet

- class Reader //used to read data sources during a scan, and control peripheral equipment.
 - class ReadableReader //reads data from a Readable object
 - class PCReader //reads from the Photon Counter, and links its gating to a Scanner
 - class LCDReader //reads from another Reader while controlling the LCD retarder

4.2 Instrument Interface

The most generic interface class describing laboratory instruments.

We use this class to address all lab equipment from a unified interface. This is not a parent class in the truest "is-a" convention, but since LabVIEW 8.6 does not include C++-style multiple inheritance nor Java-style interfaces, we simply define this "interface class" and set it as the parent of all classes that implement it.

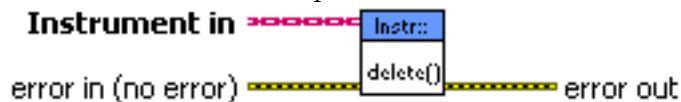
Member fields contain a name and a VISA resource for communicating with the Instrument - use this class's VISA Get and VISA Save to access it. Useful dynamic methods include Check Status and Control Panel. These methods should be universal to all laboratory equipment.

Known subinterfaces include Scannable and Readable.

4.2.1 Instrument.lvclass: _Destroy.vi

Superclass destructor: close all handles to and release all resources associated with this Instrument, then deallocate it.

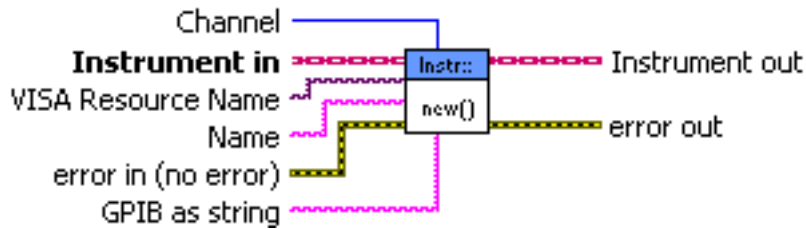
IMPORTANT: when destroying a class that inherits from Instrument, destroy your subclass first, and call the parent Instrument::_Destroy.vi LAST. This ensures your subclass destructor operates on a valid Instrument while it releases its resources.



4.2.2 Instrument.lvclass: _InitInstrument.vi

(Protected) Superclass constructor. Associate a name, and optionally a VISA Resource, with this Instrument. If Channel is specified, this constructor will append Channel to the Instrument's Name.

Call this superclass constructor as the beginning of your subclass constructor. This will ensure you always operate on a valid Instrument. To ensure your subclass constructor produces objects of the correct type, ALWAYS connect your subclass constant to "Instrument in" on this superclass constructor.



4.2.3 Instrument.lvclass:_Refresh.vi

Reconstruct the object, if needed, after deserializing it from a file. Refresh any stale resource handles. One way to accomplish this is to delete and reconstruct the object.



4.2.4 Instrument.lvclass:Check Status.vi

(Dynamic) Check if the instrument is powered on and update its member fields with the instrument's current status. Optionally return a text status message.



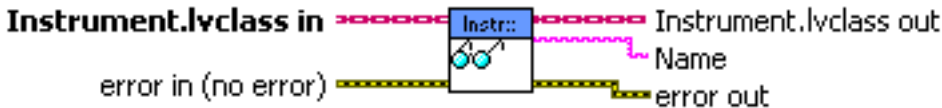
4.2.5 Instrument.lvclass:Control Panel.vi

(Dynamic) A user-level application that simulates the physical front panel of the Instrument. Provide the user with buttons to manually control features of the instrument, and provide indicators of its current state.



4.2.6 Instrument.lvclass:Read Name.vi

Return this Instrument's name from its member field.



4.2.7 Instrument.lvclass:Scan Setup.vi

(Dynamic) Send commands to configure the instrument according to the class member fields.



4.2.8 Instrument.lvclass:Setup UI.vi

Setup UI constructs an object – populates its member fields – from a front panel the user can see. The VI should be as simple as possible so that a UI loop can poll it continuously for updates.

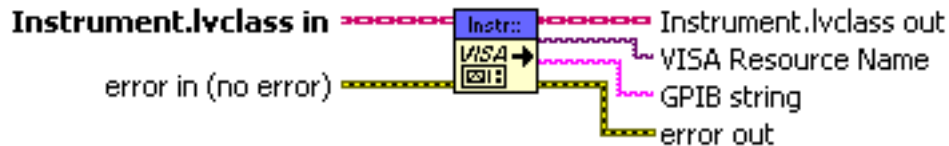
The VI is designed to be used as a subPanel within another VI. Wire true to "Set UI to input" only on the first call, just after inserting this VI into a subPanel, to populate its controls with the class's current values.

Setup UI is distinct from the class constructor. Class constructors are not polymorphic, yet configure dialogs need to configure objects without knowing their type – and polymorphism is the necessary solution. Thus, Setup UI is a polymorphic class constructor. This continuously polled "Setup UI" concept is a kludge: a better idea would be an XControl, but those require a steeper learning curve.



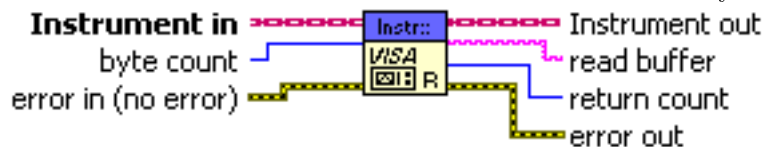
4.2.9 Instrument.lvclass:VISA Get Session.vi

(Dynamic) Access the VISA session constructed for this Instrument. (Use this to send commands to the Instrument). Override this method if the Instrument has communication idiosyncrasies.



4.2.10 Instrument.lvclass:VISA Read Async.vi

(Dynamic) Read from this Instrument using an asynchronous VISA call. Override this method if the Instrument has communication idiosyncrasies.



4.2.11 Instrument.lvclass:VISA Save Session.vi

(Dynamic) Save the VISA session constructed for this Instrument. (Use this after sending commands to the Instrument). Override this method if the Instrument has communication idiosyncrasies.



4.2.12 Instrument.lvclass:VISA Write Async.vi

(Dynamic) Write to this Instrument using an asynchronous VISA call. Override this method if the Instrument has communication idiosyncrasies.



4.3 Readable Interface

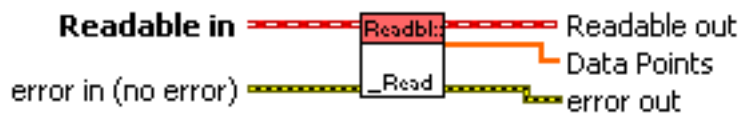
Subinterface of Instrument that represents a piece of equipment whose purpose is to perform a measurement. This class defines the Y-axis of a completed scan graph. Examples include a lock-in amplifier or a voltmeter.

This class has one substantive (protected) method - `_Read.vi`. Use "Read Data.vi" which is the public interface to this method.

This interface class is not necessarily independent of Scannable.lvclass - it might be useful to read the current position of a Scannable. However, since it is impossible to perform multiple inheritance in LabVIEW 8.6, making Scannables also Readables requires making Scannable a subclass of Readable – which leaves a mess. Just something to think about for the future.

4.3.1 Readable.lvclass: _Read.vi

(Protected, Dynamic, Pure Virtual/Abstract) Perform a measurement and return one or more channels of data. Outsiders should not call this method, but instead the public "Read Data.vi".



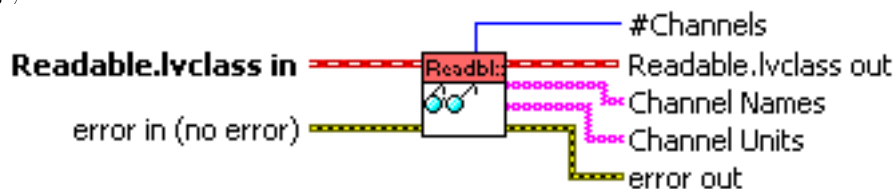
4.3.2 Readable.lvclass:Graph.vi

(Dynamic) Take a data point, then append it to an internal history buffer. Return the history buffer as a waveform array configured for graphing.



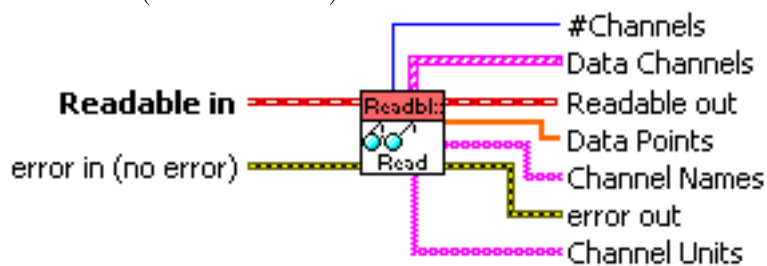
4.3.3 Readable.lvclass:Read Channel Info.vi

(Dynamic) Return information about the readable data channels this object provides, e.g., the name of each data source and its unit.



4.3.4 Readable.lvclass:Read Data.vi

Perform a measurement and return one or more channels of data. Public interface to "_Read.vi." (Use this one!)

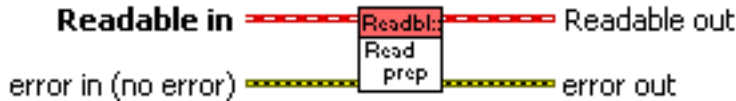


4.3.5 Readable.lvclass:Read Prep.vi

(Dynamic) Prepare to read. This is in case the Readable object needs to change some settings before the time delay occurs. One example: if the photon counter

needs to start counting, call this method and then wait until the count is complete.

The "_Read" method will then stop counting and return the total counts.



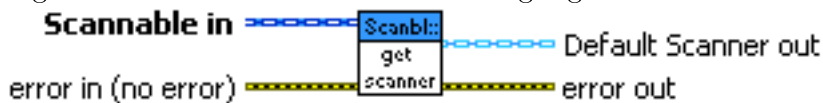
4.4 Scannable Interface

Subinterface of Instrument that describes an instrument whose purpose is to control a variable over the course of an experiment. This class defines the X-axis of a completed scan graph. Examples include a magnetic field, a spectrometer, or a pulse generator.

This subinterface class defines only one method - Get Scanner.vi. This is by design, since Scannable is modeled after Iterable from the Java language.

4.4.1 Scannable.lvclass:Get Scanner.vi

(Dynamic) Returns the default Scanner object that scans over this Scannable. (Note that a Scannable may have multiple Scanners associated with it). A Scanner is an analogue to an Iterator class in other languages.

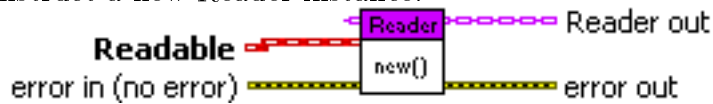


4.5 Reader Interface

The Reader class serves to decouple Scan Driver logic from the generic Instrument hierarchy. It extends the capabilities of Readable objects with scan-specific features, such as a "dwell time" to wait between Read Prep and Read. Besides that, it is mostly a wrapper around the Readable class.

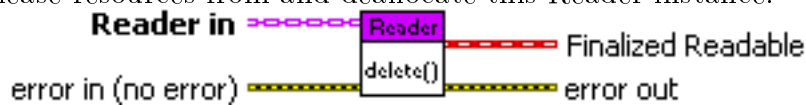
4.5.1 Reader.lvclass:Reader.vi

Construct a new Reader instance.



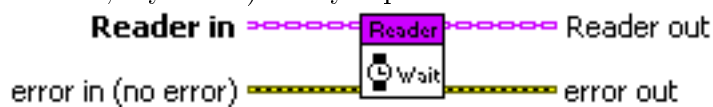
4.5.2 Reader.lvclass:_Destroy.vi

Release resources from and deallocate this Reader instance.



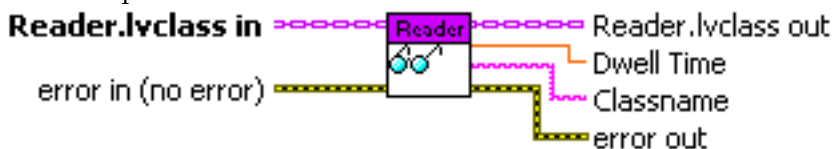
4.5.3 Reader.lvclass:Dwell Time Delay.vi

(Protected, Dynamic) Delay a predetermined time before taking a data point.



4.5.4 Reader.lvclass:Read Dwell Time.vi

Return the pre-set dwell time inside this Reader instance.



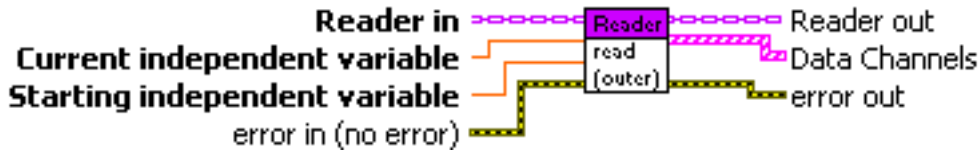
4.5.5 Reader.lvclass:Read Readable.vi

(Protected) Return the Readable object associated with this Reader instance, to allow direct manipulation of that Instrument.



4.5.6 Reader.lvclass:Read.vi

Prepare for read, wait the time delay, and take data from up to 4 channels.



4.5.7 Reader.lvclass:Scan Setup.vi

(Dynamic) Send commands to configure the instrument based on the class member fields.

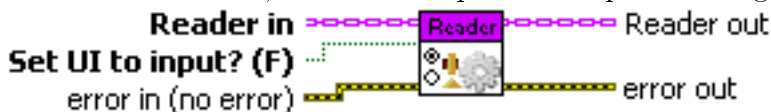


4.5.8 Reader.lvclass:Setup UI.vi

Setup UI constructs an object – populates its member fields – from a front panel the user can see. The VI should be as simple as possible so that a UI loop can poll it continuously for updates.

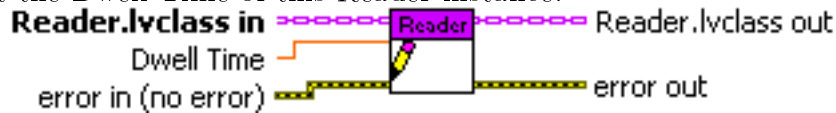
The VI is designed to be used as a subPanel within another VI. Wire true to "Set UI to input" only on the first call, just after inserting this VI into a subPanel, to populate its controls with the class's current values.

Setup UI is distinct from the class constructor. Class constructors are not polymorphic, yet configure dialogs need to configure objects without knowing their type – and polymorphism is the necessary solution. Thus, Setup UI is a polymorphic class constructor. This continuously polled "Setup UI" concept is a kludge: a better idea would be an XControl, but those require a steeper learning curve.



4.5.9 Reader.lvclass:Write Dwell Time.vi

Set the Dwell Time of this Reader instance.



4.5.10 Reader.lvclass:Write Readable.vi

(Protected) Update the Readable inside this Reader instance, after it has been accessed with "Read Readable.vi".



4.6 Scanner Interface

An object that manipulates a Scannable over the course of an experiment. Sets a range over which to scan, and controls its position at the demand of a host application. Scanner and Scannable are modeled after Iterator and Iterable from the Java language.

Note that a given Scannable may have one or more Scanners. Examples include a pulse generator with Scanners to vary either the pulse width, pulse height, or pulse delay; a magnetic field that can sweep either continuously or at discrete steps; and a spectrometer that can vary its wavelength or slit width.

4.6.1 Why Scanner and Scannable?

It probably seems silly to have two separate classes, `Scannable` and `Scanner`, especially since the former defines a single method. The reasons for this design decision are threefold:

1. Maximum flexibility in using instruments.

Some instruments have more than one possible way to “scan” over them. For example, one can vary a Magnetic Field either continuously or by careful steps. Two different objects are needed here.

2. Decouple application from interface.

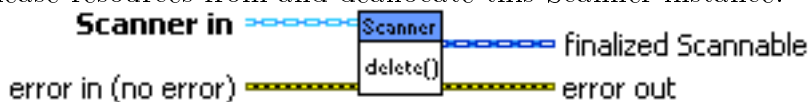
`Scanner` is an internal class meant exclusively for the Scan Driver application. It probably cannot be used well in other applications. The `Instrument` interface is intended for use in more than one application.

3. Follow pattern of other languages.

`Scanners` are meant to correspond to `Iterators` in other programming languages, while `Scannables` correspond to an `Iterable Collection`. Someday, hopefully `Scanners` will be generic enough to use outside of `Scan Driver`.

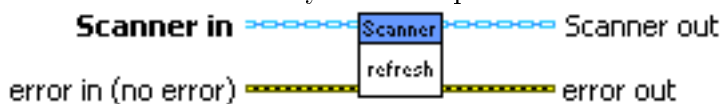
4.6.2 `Scanner.lvclass:_Destroy.vi`

Release resources from and deallocate this `Scanner` instance.



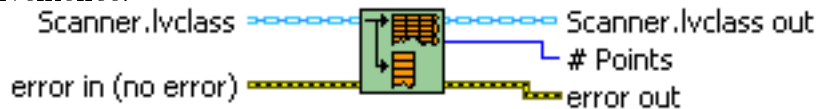
4.6.3 `Scanner.lvclass:_Refresh.vi`

Reconstruct the object, if needed, after deserializing it from a file. Refresh any stale resource handles. One way to accomplish this is to delete and reconstruct the object.



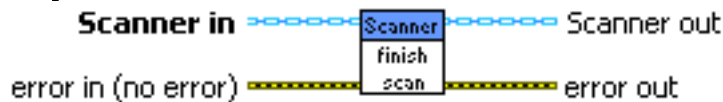
4.6.4 Scanner.lvclass:Create ArrayVarying.vi

Generate an internal array of independent variables from the Range. You do not need to use this array if it makes no sense for your subclass; it is merely provided for convenience.



4.6.5 Scanner.lvclass:Finish Scan.vi

(Dynamic) Perform any cleanup after the last step. Restore the instrument to its initial pre-scan state if "Return to Start" is true.



4.6.6 Scanner.lvclass:First Step.vi

(Dynamic) Reset the instrument and prepare to scan. Go to step #0 - the first step. If desired, the scanner may do this in a different way than usual (i.e. by preparing to move a long distance, or reset to home position).



4.6.7 Scanner.lvclass:Has Next.vi

(Dynamic) Return TRUE if this Scanner has not yet completed all steps in its pre-configured Range.



4.6.8 Scanner.lvclass:Next Step.vi

(Dynamic) Move to the next preconfigured step of this Scanner, and return the current position (actual, not intended) of this Scanner. Increment the step counter by one.



4.6.9 Scanner.lvclass:Range.ctl

Store the Range for a Scanner, in the Scanner's natural units.

xi: first point

xf: last point

dx: step size between points

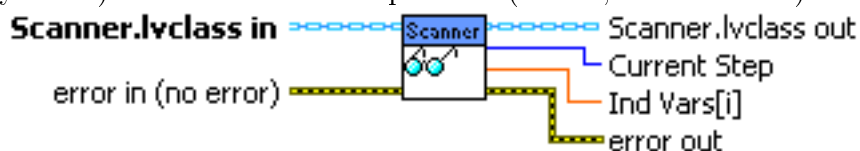
Return to start?: if you would like the application to "rewind" the scanner to its original position when it is finished

Note that only this range control gets propagated in the Scanner copy constructor - everything else is reinitialized.



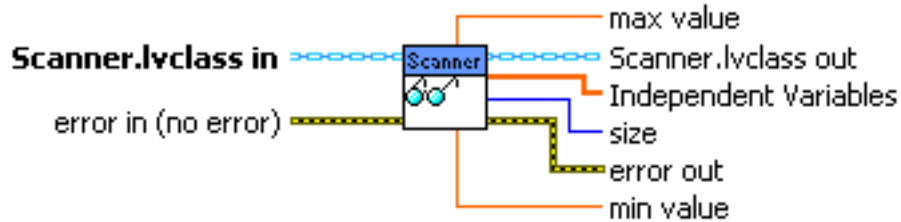
4.6.10 Scanner.lvclass:Read Current Step.vi

(Dynamic) Return the current position (actual, not intended) of this Scanner.



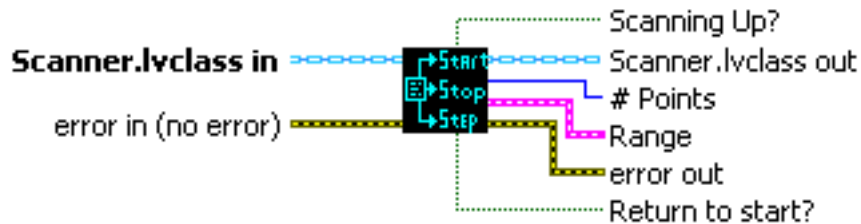
4.6.11 Scanner.lvclass:Read Independent Variables.vi

Return the internal array of independent variables for this Scanner, e.g., to write out to a file. Use this method with extreme caution, since it breaks the encapsulation for this class. Do not use it to manipulate a Scanner's internal state.



4.6.12 Scanner.lvclass:Read Range.vi

Return the Range of this Scanner.



4.6.13 Scanner.lvclass:Read Scannable.vi

(Protected) Return the Scannable object associated with this Scanner instance, to allow direct manipulation of that Instrument. Scannables are stored within Scanners, because LabVIEW does not allow inner classes. If inner classes were possible, it would be better to define Scanner as an inner class of Scannable.



4.6.14 Scanner.lvclass:Scan Setup.vi

Configure the Instrument based on the class member fields. This step comes after the user has closed the "Scan Setup" window and before the actual scan – so instruments are configured with the user's specified settings.



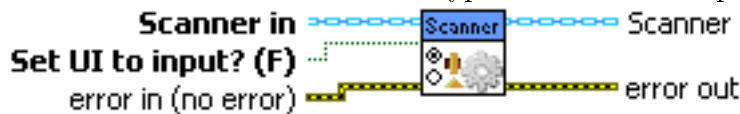
4.6.15 Scanner.lvclass:Scanner.vi

Protected superclass constructor for Scanner. Make sure you wire your class to the input of this method so it constructs objects with the correct type.



4.6.16 Scanner.lvclass:Setup UI.vi

Scanner::Setup is a constructor that has no parameters and faces the user. The Factory calls this VI to let the user configure this Scanner. It is dynamically dispatched in order to let the user choose the type as well as the options of the scanner.



4.6.17 Scanner.lvclass:Status.vi

(Dynamic) Return a text status message reflecting the current state of this Scanner.



4.6.18 Scanner.lvclass:Write Range UI.vi

A Setup UI method for the superclass "Range" of scanner.



4.6.19 Scanner.lvclass:Write Range.vi

Set the Range of this Scanner to the Range of another Scanner.



4.6.20 Scanner.lvclass:Write Scannable.vi

(Protected) Update the Scannable inside this Scanner instance, after it has been accessed with "Read Scannable.vi".



4.7 Scan Driver Class

Scan Driver is a simple application that configures arbitrary Scannable and Readable Instruments and then scans over them to generate a graph of data. It interfaces with all major classes in this project: Instrument, Scannable, Readable, Scanner, Reader, and File IO.

File IO is a friend of Scan Driver so that it can access member fields and write them to a file.

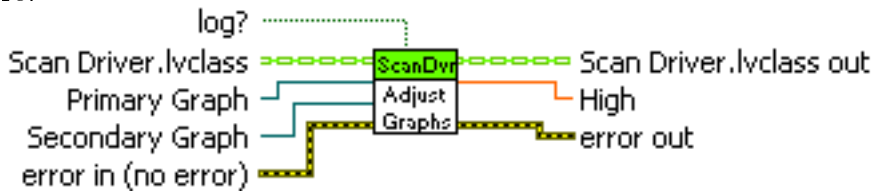
4.7.1 Scan Driver.lvclass:_Destroy.vi

Release resources from and deallocate this Scan Driver instance.



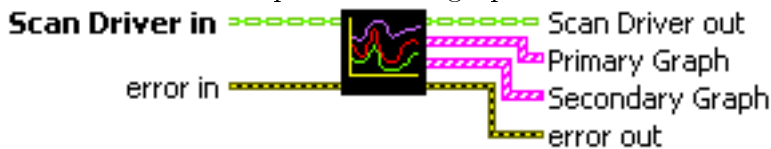
4.7.2 Scan Driver.lvclass:Adjust Min-Max.vi

Set the boundaries of the onscreen scan graphs according to the extrema of the data store.



4.7.3 Scan Driver.lvclass:Build Scan Graph.vi

Build and return the updated scan graph from the data store.



4.7.4 Scan Driver.lvclass:Create Data Store.vi

Initialize the internal Data Store. Call member Scanner instance to do the same.



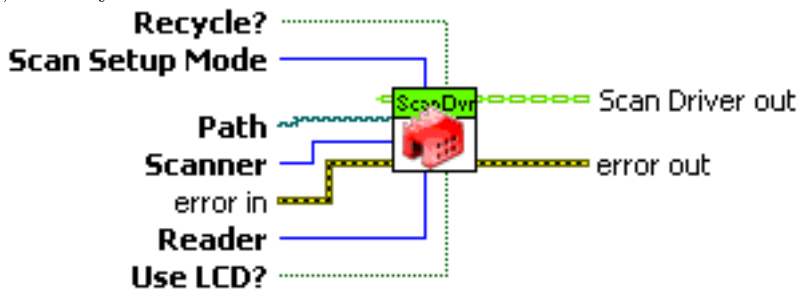
4.7.5 Scan Driver.lvclass:Estimate Time.vi

Estimate the time needed to complete this Scan Driver instance, using data from member Scanner and Reader.



4.7.6 Scan Driver.lvclass:Factory.vi

Construct a new Scan Driver instance using the given parameters, load one from a file, or recycle an old one.



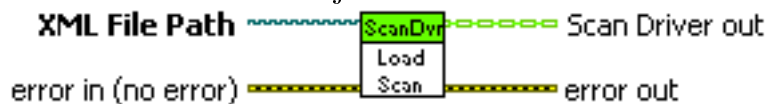
4.7.7 Scan Driver.lvclass:Finish Scan.vi

Reset the scan and all Scanners and Readers. Write the data file. Prepare to scan again if more than one scan is requested.



4.7.8 Scan Driver.lvclass:Load Scan.vi

Unflattens a Scan Driver instance from the specified XML file. If an error occurs, returns the class default object.



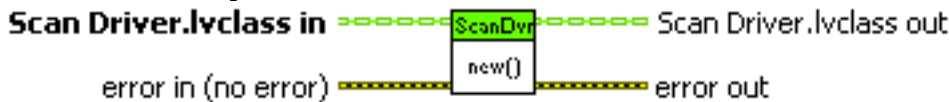
4.7.9 Scan Driver.lvclass:Pre-Scan Check.vi

Call "Scan Setup" on all Instruments contained in this Scan Driver instance. This will configure the Instruments according to their member fields. Move the Scanner to the First Step.



4.7.10 Scan Driver.lvclass:Scan Driver (for Setup UI).vi

Constructor. Finish construction of the Scan Driver after running the Setup UI methods on its component classes.



4.7.11 Scan Driver.lvclass:Scan.vi

Main method for Scan Driver. Scan over the given Scanner, read data points from the given Readable, generate an onscreen graph, and write a data file.



4.7.12 Scan Driver.lvclass:Settings Panel.vi

Main method of the Scan Driver class. Construct a new Scan Driver instance from the given options, or load and deserialize one from an XML file, or modify the last run scan. Configure the instance. Run the scan, and repeat it if more than one scan is requested.



4.7.13 Scan Driver.lvclass:Setup.vi

GUI constructor for new Scan Driver instances. Call the component objects' Setup UI methods and use their GUIs to construct them too.



4.8 Creating New Objects

To create new objects that are compatible with the existing software, one needs only to implement these interfaces. Here is the procedure:

1. Open up the Main.lvproj. Stop any running programs.
2. Determine if your new equipment fits under Scannable or Readable, or as a generic Instrument.
3. Create a new folder to hold your code files.
4. Create a new class for your equipment. Refer to the LabVIEW documentation to see how this is done. (Currently: create a new folder, right-click on it in the Project Explorer, and select New... Class).
5. Set the class to inherit from the appropriate parent class, e.g. Scannable, Readable, etc. Refer to the LabVIEW documentation. (Currently: right-click on the .lvclass entry in the Project Explorer, select Properties, and select Inheritance).
6. Save the new .lvclass file.
7. Carefully study the interface (public methods) of this parent class and its documentation.

8. Carefully study other existing child classes of this parent class. Study how they work. Determine where and why child classes overrode default parent methods, as this may aid you in determining which methods to override in your class.
9. Implement your class's code. Create its own class-specific methods and override default parent methods where appropriate. To override a method, consult the LabVIEW documentation. (Currently: right-click on the .lvclass in the Project Explorer and select New... VI for Override).
10. In application programs which use objects, locate code points where the user selects which object to construct. Add your object to the list as appropriate.
11. See also: the LabVIEW user guide under the "Creating LabVIEW Classes" topic, or the on-line user guide at http://zone.ni.com/reference/en-XX/help/371361F-01/lvconcepts/creating_classes/.

Chapter 5

Applications

This chapter describes the culmination of this project: combining and utilizing the object classes to create applications. We will discuss two such applications, Etch-A-Sketch and Scan Driver. These applications are freely available, along with all other code, at <http://coltonlab.googlecode.com> (as of this writing).

5.1 Etch-A-Sketch

Etch-A-Sketch is a simple application that graphs data onscreen in real-time, much like a laboratory oscilloscope. Etch-A-Sketch reads data from any **Readable Instrument**. Adding new data sources is as easy as creating a new **Readable** (section 5.2.1).

The user selects which graphs to display from a list of preconfigured objects on the front panel (figure 5.1), or can easily add another to the list by modifying the block diagram (figure 5.2). The main read loop in Etch-A-Sketch (figure 5.3) uses a precise timer to evenly demarcate read commands. The data then scrolls across the screen in real-time. This application is useful to align optics, trace signal paths, and verify that detectors are functioning correctly.

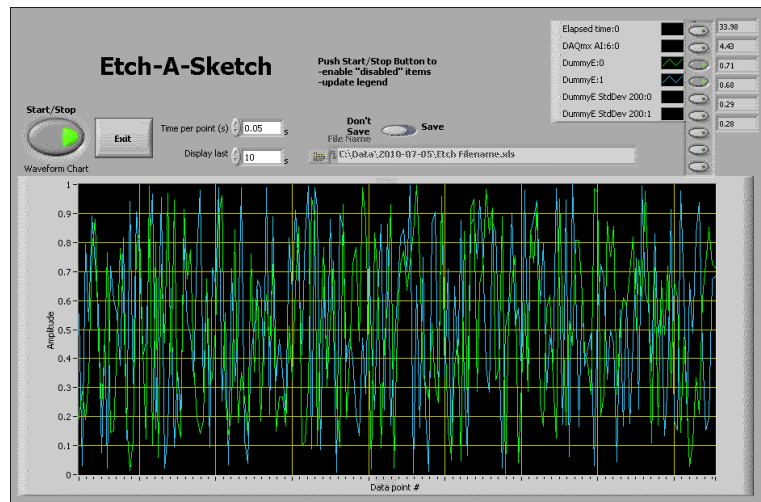


Figure 5.1 Etch-A-Sketch front panel.

The `Readable` class extends itself easily to the “pseudo-Instruments” `Minus` and `StdDev`. These two classes take `Readables` as parameters, and output difference and standard deviation signals, respectively. This allows extremely flexible configurations of data sources within the program, as demonstrated in figure 5.2.

5.2 Scan Driver

Scan Driver is the main application program of the Colton Lab Scan Software. Its purpose is to control multiple instruments to perform an experiment, and generate an x-y graph of the results. Scan Driver is discussed in sections 3.1.1 and 4.7.

The user begins by constructing new scan objects using the drop-down menus (figure 5.4). The items listed correspond to `Scanner` and `Reader` objects described earlier.

The user may also choose to load a saved scan file. Saved scan files are simply the member fields of all the various objects serialized into XML format (see section 3.2.2). LabVIEW internally handles the ABI versioning and dynamic serialization/deserialization (figure 5.5).

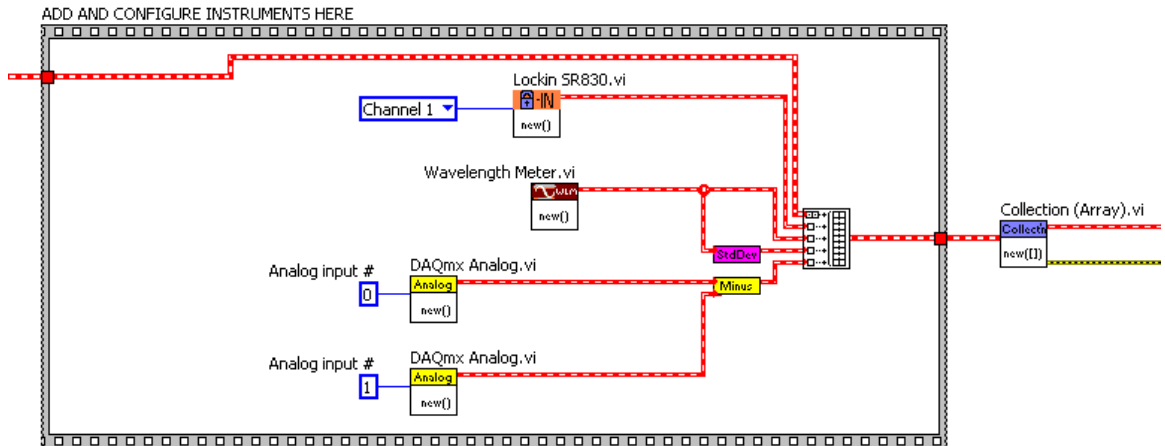


Figure 5.2 Constructing Objects in Etch-A-Sketch. Two “pseudo-Instruments” are present: StdDev and Minus.

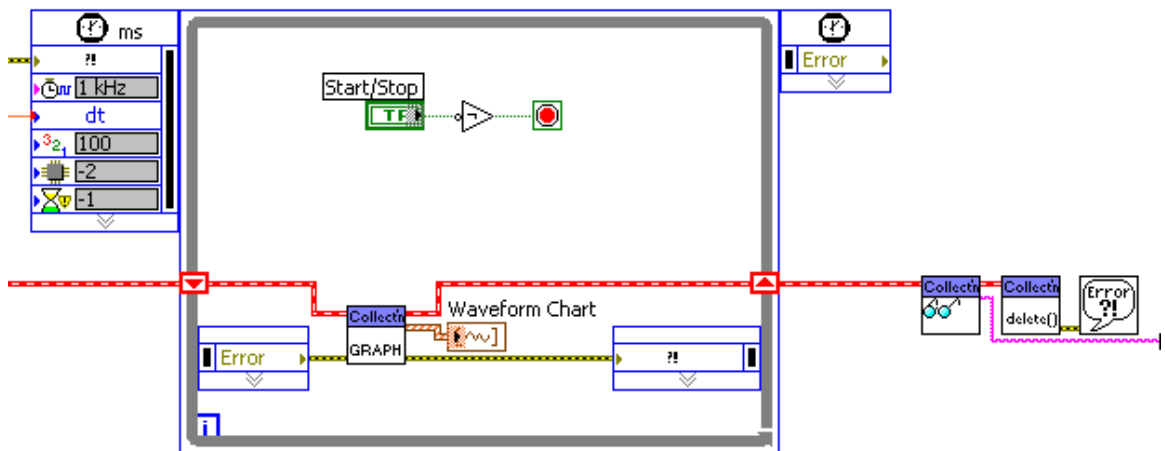


Figure 5.3 Main read loop in Etch-A-Sketch.

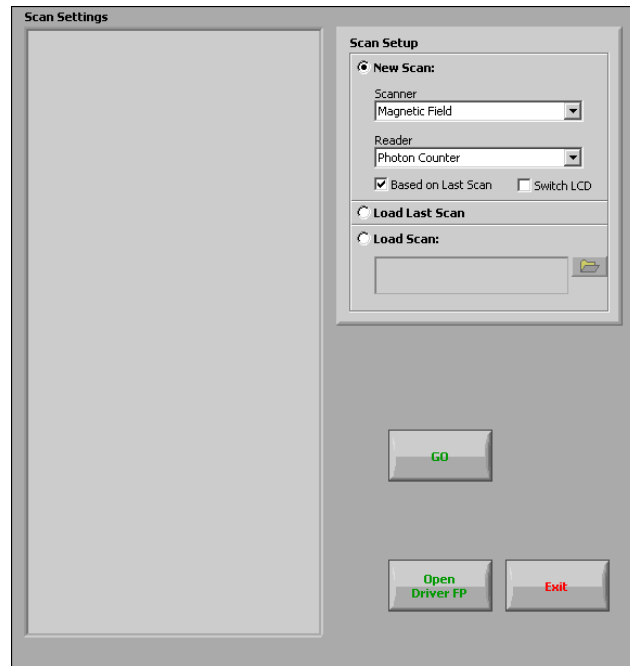


Figure 5.4 Scan Driver constructor front panel.

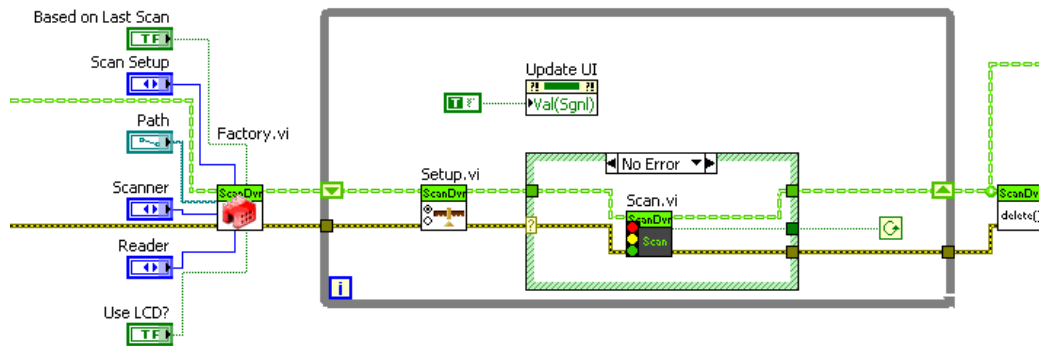


Figure 5.5 Scan Driver constructor block diagram.

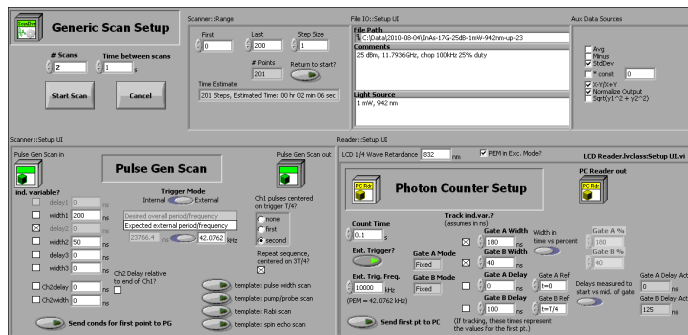


Figure 5.6 Scan Driver setup UI front panel.

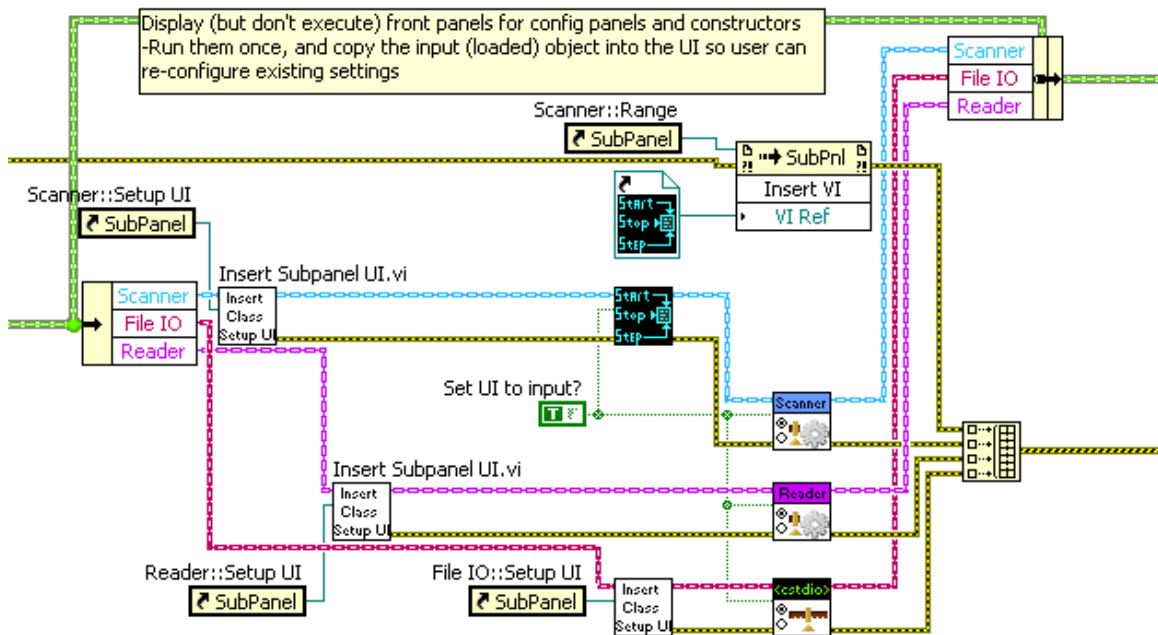


Figure 5.7 Setup UI block diagram.

Once a new scan is constructed, it is passed to the Setup UI method (figure 5.6). This allows the user to configure the vast array of scan settings. The recessed areas on the front panel represent “sub panels,” a way to snap in user interfaces into the same window. These interfaces are also called Setup UI and each object defines its own Setup UI method. This is simpler than implementing event handler callbacks through LabVIEW’s XControl facility. The following code (figure 5.7) enables the front panel to dynamically snap-in these interfaces.

Once the user configures all settings, the scan begins. The front panel displays progress and and displays the collected data (figure 5.8).

The main loop does the following:

1. Calls `Get Next Step` on the `Scanner`
2. Reads the data from the `Reader`
3. Post-processes and displays the data

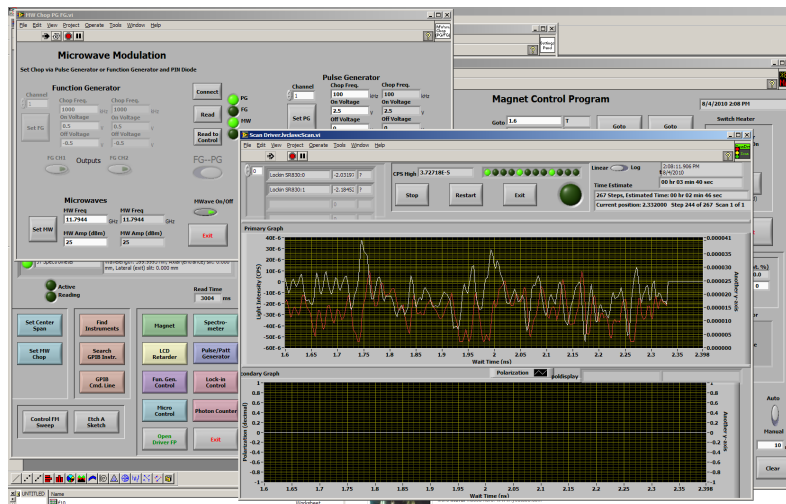


Figure 5.8 Scan method front panel.

4. Calls Has Next on the Scanner

After the scan is complete, the user returns to the constructor menu and may scan again using the same settings if desired.

5.2.1 How to Expand Scan Driver

Scan Driver can easily expand to use new equipment through the creation of new object classes.

To create a new `Readable` object, simply define the object class and override its `_Read` method. Then modify menus where appropriate to allow the user to choose and construct your new object.

To create a new `Scanner`:

1. Create a `Scannable Instrument` object, and its own directory. Set the object to inherit from `Scannable.lvcClass` as outlined in section 4.8.
2. Create member fields in this object that deals with instrument settings, e.g. current position, display data in meters or feet, sample rate, etc.

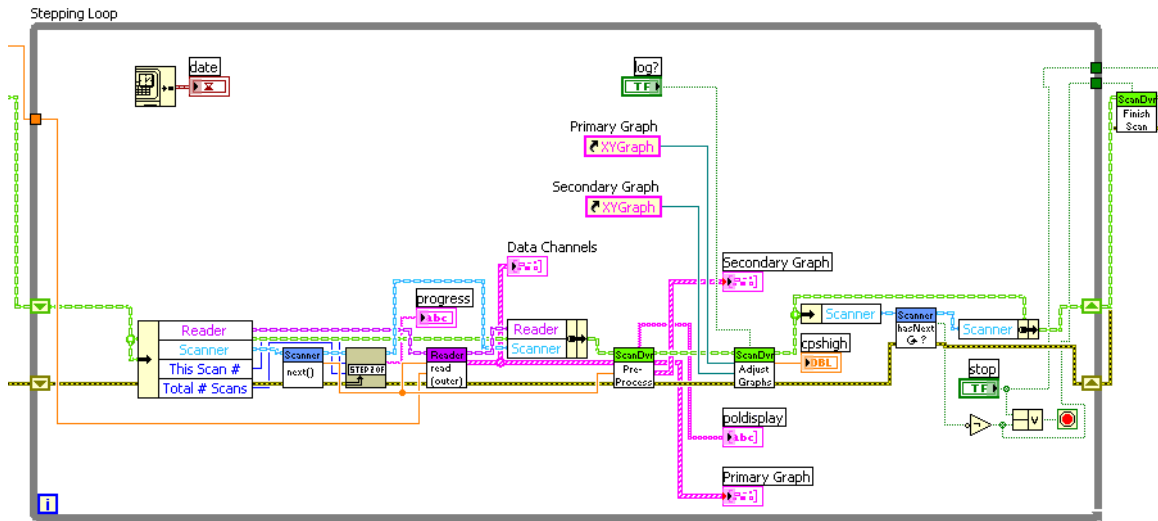


Figure 5.9 Scan method main loop block diagram.

3. Override methods for generic instrument behavior, e.g. Check Status.
4. Implement a Control Panel method if desired. Override the Control Panel from the parent Instrument class as outlined in section 4.8.
5. Create a Scanner object, and its own directory. Set the object to inherit from Scanner.lvclass. Again, follow the procedure from section 4.8.
6. Create member fields in the Scanner object that deals with user scan settings, e.g. scan rate, scan up or scan down, pulse output during scan.
7. Review the methods in the Scanner class's interface. Study their behavior.
8. Override methods from the parent Scanner class where you do not wish to use the default behavior. Implement your own methods. Commonly overridden methods include: First Step, Next Step, Scan Setup, Finish Scan.
9. Modify menus where appropriate to allow the user to select and construct your new Scanner object.

Chapter 6

Conclusion

Many of the benefits of the object-oriented system have already been realized:

- Source code revision control (section 3.3.1) made possible this project's sweeping changes without disrupting daily research.
- New features and conveniences are now possible. In particular, the load/save serialization feature (section 3.2.2) has already proven useful to create "template" experiment sessions where common parameters are already set.
- The object-oriented framework has enabled rapid development of new types of experiments, such as time-resolved Kerr rotation (section 1.2.3.2), that were not possible with the previous version of the software.
- Other labs at BYU have been able to reuse our software, and because the code is freely available, a few Open Source "aggregator" web sites have already begun to redistribute the code as of this writing.

The true evaluation for any project is the test of time. Not all requirements are known beforehand and not every problem can be foreseen. All designs are iterative,

and this project is simply one of many steps to follow. There are inevitably known and unknown bugs in this implementation, and someday it too will be thrown out.

The author hopes that this design is flexible and robust enough to continue to meet the needs of the laboratory for many years into the future. And when it again comes time to redesign, the author hopes that the lessons learned from this design can influence the next.

Bibliography

- [1] F. Meier and B. P. Zakharchenya. *Optical Orientation*. North-Holland, Amsterdam (1984).
- [2] For an overview of spintronics in semiconductors, see D.D. Awschalom, D. Loss, N. Samarth, *Semiconductor Spintronics and Quantum Computation*, Springer (2002).
- [3] B. Heaton, J.S. Colton, D.N. Jenson, M.J. Johnson, A.S. Bracker. “Nuclear effects in Kerr rotation-detected magnetic resonance of electrons in GaAs.” *Solid State Communications* 150 (2010), 244-247.
- [4] J. S. Colton, T. A. Kennedy, A. S. Bracker, and D. Gammon. “Microsecond spin-flip times in n-GaAs measured by time-resolved polarization of photoluminescence.” *Phys. Rev. B* 69, 121307 (2004).
- [5] J.S. Colton, T.A. Kennedy, A.S. Bracker, D. Gammon, and J. Miller. “Optically oriented and detected electron spin resonance in a lightly doped n-GaAs layer.” *Phys. Rev. B* 67, 165315 (2003).
- [6] J.S. Colton, T.A. Kennedy, A.S. Bracker, D. Gammon, and J. Miller. “Dependence of optically oriented and detected electron spin resonance on donor concentration in n-GaAs.” *Solid State Comm* 132, 613 (2004).

- [7] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press; 2nd edition (June 9, 2004).
- [8] Stephen Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math; 7th edition (June 29, 2006).
- [9] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. Wiley; 4th edition (August 24, 2005).
- [10] *LabVIEW 8.6 User Guide*. National Instruments (June 2008).

Glossary

| | |
|--------------------------|--|
| abstraction layers | hierarchical groupings of a completed framework, page 24 |
| child classes | define new and additional behavior from a parent class using inheritance and overriding, page 18 |
| class | defines an object in abstract terms; the object is its physical representation (instance), page 16 |
| Colton Lab Scan Software | main software application that is the focus of this project, used to study spin dynamics, page 11 |
| conduction band | the lowest energy band in a solid which is normally completely empty at room temperatures, page 2 |
| data structure | a computer's internal representation of an object, page 16 |
| decomposition | method of planning a software project by splitting goals into tasks and subtasks, page 22 |
| degenerate | different electron states that have the same energy, page 4 |
| dynamically dispatched | a polymorphic method; see polymorphism, page 20 |
| electron spin resonance | experiment to induce transitions between spin states by applying microwave energy at a resonant frequency, |

| | |
|----------------------|--|
| | page 4 |
| encapsulation | a way of hiding internal complexity to make software components more robust and easier to integrate, page 17 |
| field | data stored inside an object, page 16 |
| framework | collection of interfaces and interchangeable components that enables flexible application development, page 24 |
| implement | to write code that performs a pre-specified behavior, e.g. to write an interface's required methods, page 17 |
| inheritance | a child class will automatically re-use its parent's methods unless it overrides them, page 18 |
| instance | an object; the data in memory that corresponds to a class, page 16 |
| interface | allows a program to use multiple classes interchangeably as long as each defines the same methods, page 17 |
| method | function of code that manipulates an object, page 15 |
| multiple inheritance | ability to use more than one interface, page 23 |
| mutex | mutual exclusion lock; a key to the restroom: only one person can have the key at a time, page 26 |
| object | an imaginary representation of data inside a computer program, page 15 |
| optical pumping | can spin-polarize a material into either CB spin state, page 4 |

| | |
|--------------------|--|
| override | to redefine a parent class's methods with new behavior, page 18 |
| photoluminescence | photons emitted from a semiconductor material when its electrons transition from an excited state, page 2 |
| polymorphism | compiler construct that automatically chooses the method appropriate for the object when the program runs, page 19 |
| population | number of electrons in one spin state relative to another state, page 3 |
| private methods | methods that outsiders may not call; only for the class's internal use, page 17 |
| public methods | methods that a class allows others to use, page 17 |
| serialize | to turn structured data into sequential data, e.g. for writing objects to disk, page 27 |
| specific interface | methods defined for a class's own specialized purpose, page 17 |
| spin | angular momentum of an electron or photon, page 3 |
| spin-polarized | most of a material's electrons are in the same spin state, page 3 |
| strong layering | one layer may only interface with its neighbors above and below, page 24 |
| thermal effects | can spin-polarize a material if the spin-splitting energy is comparable to ambient thermal energy, page 6 |

- time-resolved Kerr rotation experiment to measure the transverse lifetime T_2^* of a material's spin states, page 10
- tree hierarchical data structure, page 27
- valence band the highest energy band in a solid which is normally completely filled at room temperatures, page 2